A Verified Compiler for an Impure Functional Language

Adam Chlipala
Harvard University
POPL 2010

What are the engineering principles that make **compiler verification** worth doing **in the real world**?

In particular, for **higher-order languages**, which have tricky binder issues

From Mini-ML to Assembly

Source language

```
e ::= c | e = e | x | e e | fix f(x). e
    | let x = e in e | ()
    | (e, e) | fst(e) | snd(e) | inl(e) | inr(e)
    | case e of inl(x) => e | inr(x) => e
    | ref(e) | !e | e := e
    | raise(e) | e handle x => e
```

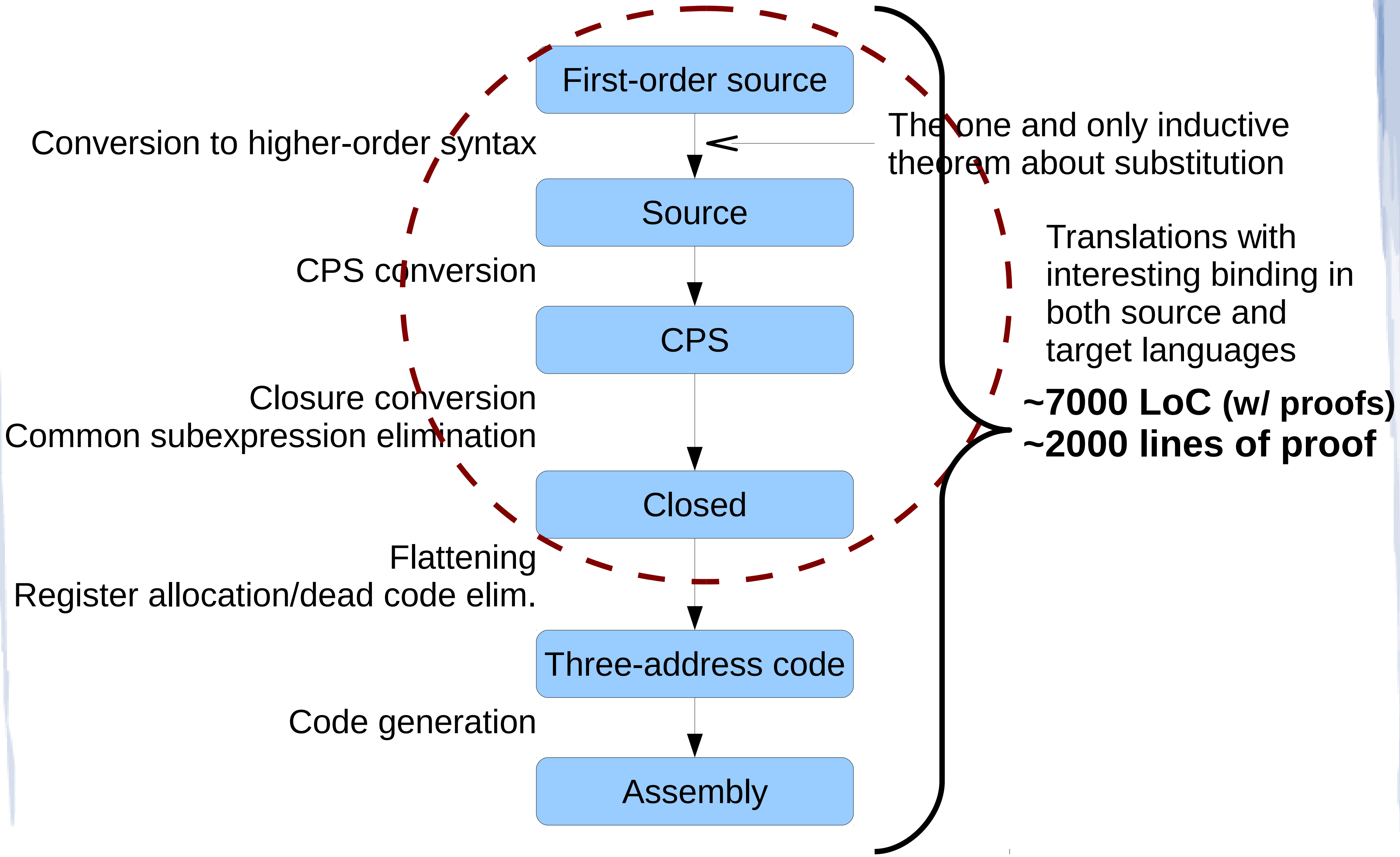
Target language

```
     Lvalues L ::= r | [r + n] | [n]
     Rvalues R ::= n | r | [r + n] | [n]
Instructions I ::= L := R | L := R == R | r += n
                 | jnz R, n
       Jumps J ::= halt | fail | jmp R
Basic blocks B ::= (I*, J)
    Programs P ::= (B*, B)
```

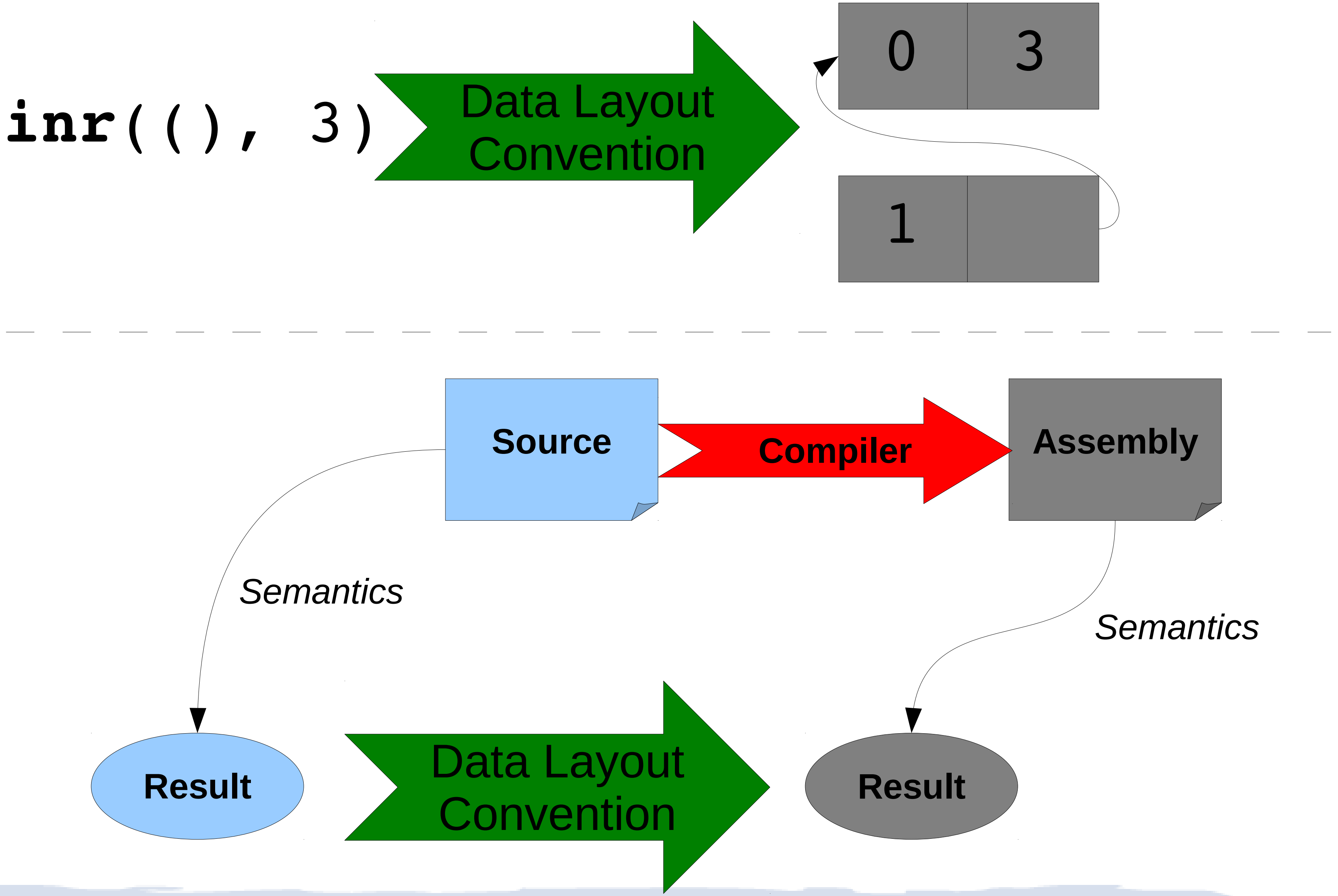
Two Main Ideas

- It's possible to *encode syntax and semantics* in a way that **avoids all auxiliary operations and lemmas about variables**.
- **Proofs** about this encoding can be **automated** effectively enough that it is not hard to **evolve a compiler and its proof over time**.

Phase Structure



Overall Compiler Correctness



Operational Semantics

`(λx. e) v` ⇒ `[x/v]e`

To verify **`compile`**, need to prove:

`compile``([x/e2]e1) =`
`[x/`**`compile`**`(e2)]`**`compile`**`(e1)`

Hiding Substitution?

`(λx. x) 1`

Encode

(Higher-Order Abstract Syntax)

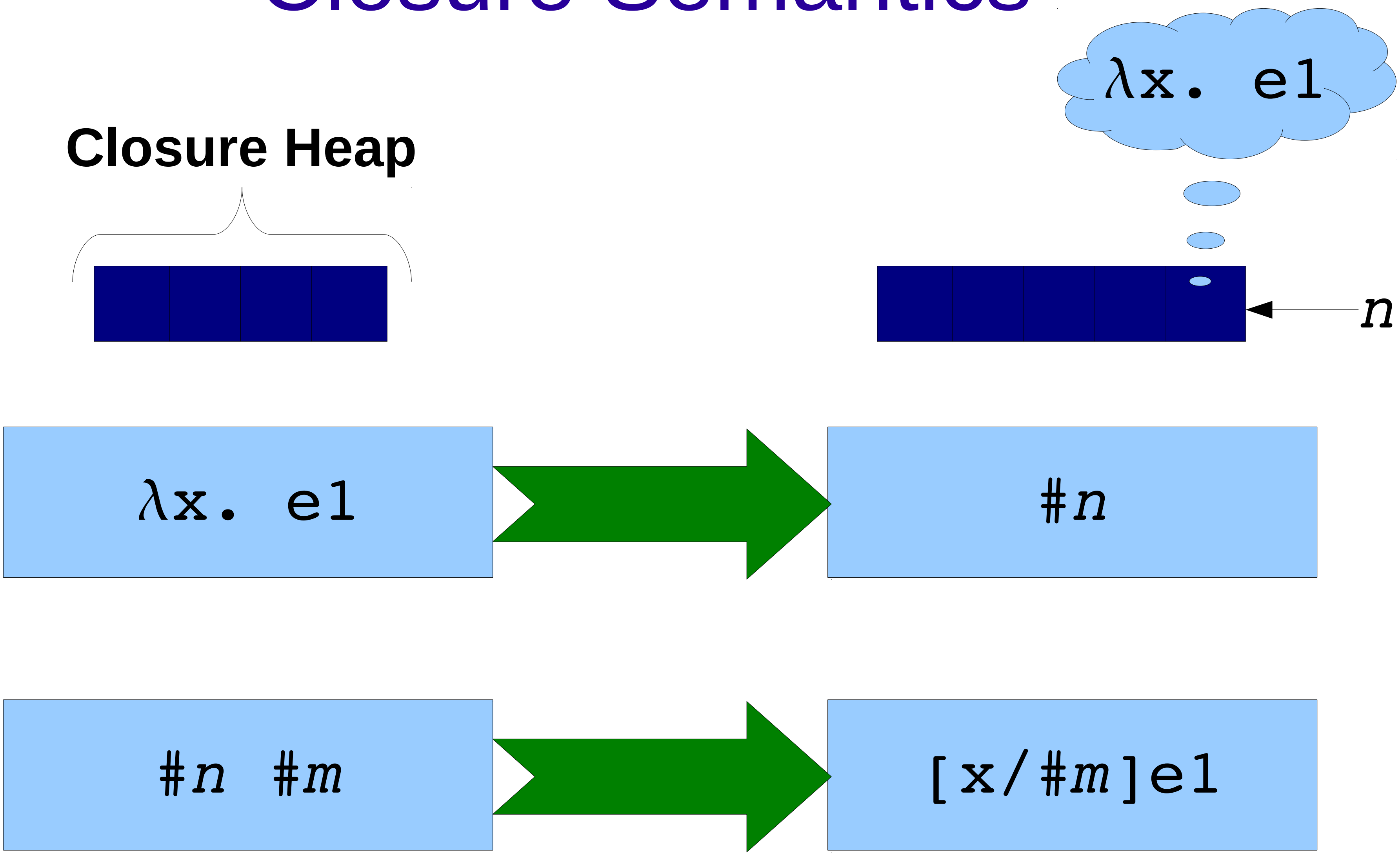
`App (Lam (fn x => x)) (Const 1)`

`App (Lam f) v` ⇒ <u>`f(v)`</u>

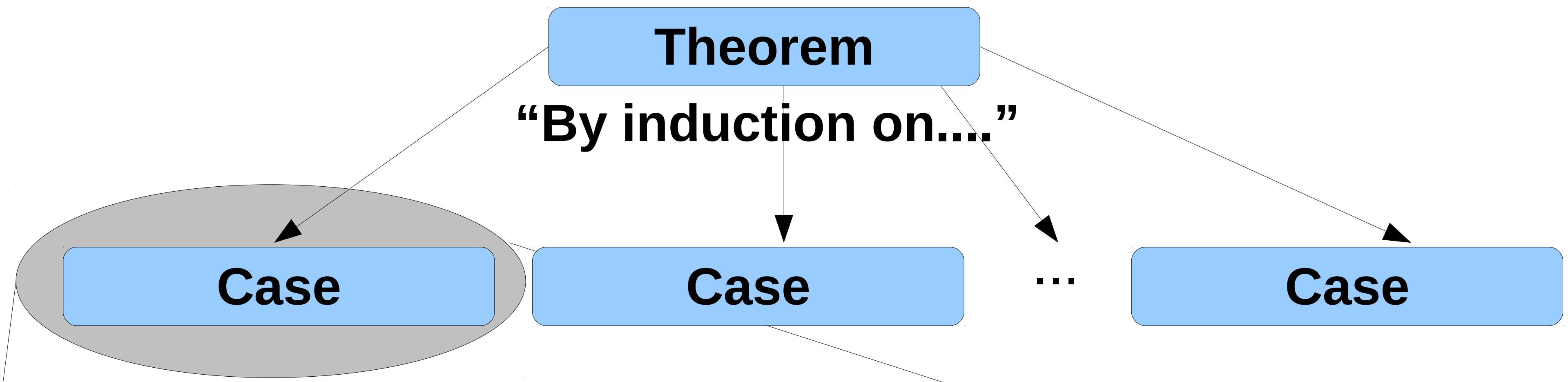
No explicit substitution!

Adding HOAS to general-purpose proof assistants creates **unsoundness**!

Closure Semantics



Automating Proofs



- Propositional simplification, partial evaluation, rewriting, ...
- Perform all useful **inversions** on hypotheses.
- **Choose IHes to instantiate** with unification variables.
- Finish with **higher-order logic programming** over rules of operational semantics and a few additional lemmas.

Proof Script Re-use

Lines of code added or changed to add new language features

	Definitions	**Theorems & Proofs**	**Time**
`let`	30	**0**	½ hour
Constants & =	150	**10**	½ day
`fix`	70	350	1 day

Almost all has to do with a new binding pattern, not the semantics of **`fix`**.

Code available in the latest **Lambda Tamer** distribution:

`http://ltamer.sourceforge.net/`

Backup Slides

Manipulating Binders

Which variables does the new expression mention?
Are they available in scope?

```
let x = ... in
let y = ... x ... in
let u = ... x ... in
let z = ... x ... y ... in
... z ... u ...
```

Does the new binding shadow a variable needed here?

De Bruijn Indices

Exactly which variables does this expression expect?

let *x* = ... **in**

let *y* = ... *0* ... **in**

let *u* = ... *1* ... **in**

let *z* = ... ~~*1*~~ *2* ... ~~*0*~~ *1* ... **in**

... *0* ... *1* ...

Did we adjust this index properly?

Higher-Order Syntax

```
let (...) (λx.
let (... x ...) (λy.
let (... x ...) (λu.
let (... x ... y ...) (λz.
... z ... u ... ))))
```

Weak Higher-Order Syntax

```
let (...) (λx : var.
let (... #x ...) (λy : var.
let (... #x ...) (λu : var.
let (... #x ... #y ...) (λz.
... #z ... #u ...
```

Parametric Higher-Order Syntax

$\forall$ **var**:
let (...) (λx : **var**.
let (... #x ...) (λy : **var**.
let (... #x ...) (λu : **var**.
let (... #x ... #y ...) (λz.
... #z ... #u ...

A piece of syntax is a first-class polymorphic function.